

# Algoritmos y Estructuras de Datos II

## Segundo parcial – Miércoles 2 de Noviembre de 2016

### Aclaraciones

- El parcial es a libro abierto.
- Cada ejercicio debe entregarse en hojas separadas.
- Incluir en cada hoja el número de orden asignado, número de hoja, apellido y nombre.
- Al entregar el parcial, completar el resto de las columnas en la planilla.
- Cada ejercicio se calificará con Promocionado, Aprobado, Regular, o Insuficiente.
- El parcial completo está aprobado si el primer ejercicio tiene al menos A, y entre los ejercicios 2 y 3 hay al menos una A. Para más detalles, ver "Información sobre la cursada" en el sitio Web.

### Ej. 1. Diseño

Un investigador escribe varios artículos simultáneamente. Cada artículo está compuesto por secciones, que no necesariamente son escritas en orden. El investigador puede realizar modificaciones al texto de alguna sección cuando lo considere necesario, lo cual genera una nueva versión. También puede arrepentirse de sus modificaciones y volver a alguna de las versiones anteriores de la sección (generando igual una nueva versión). Para simplificar, supondremos que, salvo por esto, no es necesario poder consultar las versiones de las secciones que no sean la última (es decir, no es necesario implementar la operación VerVersion).

La revista en la que publica el investigador limita el nombre de los artículos a no más de 90 caracteres. Además el investigador suele equivocarse, y es muy probable que si realizó un cambio en una sección de un artículo, el cambio siguiente en el mismo artículo vuelva a ser en la misma sección.

La especificación es la siguiente:

SECCIÓN ES NAT, ARTÍCULO ES STRING[90], TEXTO ES SECU(CHAR)

**TAD MANEJADOR DE ARTÍCULOS**

#### observadores básicos

ExisteArtículo	: ma × artículo	→ bool	
ExisteSección	: ma n × artículo a × sección	→ bool	{ExisteArtículo(n, a)}
Versiones	: ma n × artículo a × sección s	→ nat	{ExisteArtículo(n, a) ∧ <sub>L</sub> ExisteSección(n, a, s)}
VerTexto	: ma n × artículo a × sección s	→ texto	{ExisteArtículo(n, a) ∧ <sub>L</sub> ExisteSección(n, a, s)}
VerVersion	: ma n × artículo a × sección s	→ texto	
	× nat v		{ExisteArtículo(n, a) ∧ <sub>L</sub> ExisteSección(n, a, s) ∧ <sub>L</sub> v ≤ Versiones(n, a, s)}

#### generadores

Inicio	:		→ ma	
AgregarArtículo	: ma n × artículo a		→ ma	{¬ExisteArtículo(n, a)}
AgregarSección	: ma n × artículo a × sección s × texto		→ ma	{ExisteArtículo(n, a) ∧ <sub>L</sub> ¬ExisteSección(n, a, s)}
ModificarSección	: ma n × artículo a × sección s × texto		→ ma	{ExisteArtículo(n, a) ∧ <sub>L</sub> ExisteSección(n, a, s)}
Arrepentirse	: ma n × artículo a × sección s × nat v		→ ma	{ExisteArtículo(n, a) ∧ <sub>L</sub> ExisteSección(n, a, s) ∧ <sub>L</sub> v < Versiones(n, a, s)}

#### axiomas

...

**Fin TAD**

Se debe realizar un diseño que cumpla con los siguientes órdenes de complejidad temporal en el peor caso:

- AgregarArtículo y ExisteArtículo:  $O(1)$
- ExisteSección y Versiones:  $O(\log c)$
- AgregarSección:  $O(\log c)$  donde  $c$  es la cantidad de secciones del artículo al que se agrega.
- ModificarSección y VerTexto:  $O(1)$  si es la última sección modificada de ese artículo y  $O(\log c)$  en cualquier otro caso.

Se pide:

- A) i) Escriba la estructura de representación del módulo "Manejador de artículos". No se pide diseñar todos los módulos de la estructura sino solamente éste. Describa en castellano el resto de los módulos necesarios, incluidas las estructuras auxiliares que utilice en los algoritmos de la parte B.

- ii) Escriba el invariante de representación de manera formal (usando funciones de TADs y lógica de primer orden) y también en castellano.
- B) Escriba el algoritmo de **ModificarSección** y de toda otra función auxiliar que utilice para ésta, y justifique el cumplimiento de los órdenes solicitados. Para el resto de las funciones, descríbalas en castellano, justificando por qué se cumple el orden pedido con la estructura elegida.

## Ej. 2. Ordenamiento

Un sistema de monitoreo de un proceso físico arroja periódicamente mediciones donde cada una consiste en un número natural, con  $m_t$  a la medición ocurrida en el instante de tiempo  $t$ .

Se conoce, además, el valor de los umbrales  $L$  y  $H$  con  $L < H$  tales que, cuando ocurre una medición  $m_{t_0} < L$ , vale que  $(\forall t < t_0 \wedge m_t < L \Rightarrow m_{t_0} < m_t)$  y, respectivamente, cuando  $m_{t_0} > H$ , vale que  $(\forall t < t_0 \wedge m_t > H \Rightarrow m_t < m_{t_0})$ . Dicho de otra manera, todos los valores menores a  $L$  aparecen en orden decreciente y todos los valores mayores a  $H$  aparecen en orden creciente.

Ejemplo: La secuencia  $S = [3, 17, 2, 20, 1, 23, 5, 11, 9]$  es válida para  $L = 4$  y  $H = 12$ , pues los valores menores a 4 aparecen en orden 3, 2, 1 y los valores mayores a 12 aparecen en orden 17, 20, 23.

- a) Proponga un algoritmo de ordenamiento  $\text{ordenarMediciones}(A : \text{arreglo}(\text{nat}), L : \text{nat}, H : \text{nat})$  de complejidad  $O(n)$ , suponiendo que, de  $n$  mediciones observadas,  $\log n$  de éstas caen dentro del intervalo  $[L, H]$ .
- b) Justifique detalladamente la correctitud del algoritmo y su complejidad temporal.

## Ej. 3. Dividir y Conquistar

Se dice que una matriz  $A$  de dimensión  $n$  es *pirámide invertida* si, para alguna coordenada  $(i, j)$  de la matriz (con  $i, j \in [1 \dots n]$ ) se cumple que:

i,j	1	2	3	4	5
1	20	16	9	5	11
2	16	7	4	3	6
3	9	3	1	0	3
4	11	7	5	4	7
5	12	10	8	6	9

Figura 1: Pirámide invertida con fondo en  $i=3, j=4$ .

- i)  $A_{ij} = 0$ ;
- ii) Toda fila  $k$  tiene valores decrecientes hasta la columna  $j$  y crecientes desde ésta en adelante;
- iii) Toda columna  $k$  tiene valores decrecientes hasta la fila  $i$  y crecientes desde ésta en adelante.

En otras palabras,  $A$  es una pirámide invertida si tiene una celda  $(i, j)$  que vale 0, y el resto de las celdas tienen valores estrictamente crecientes a medida que me alejo en cualquier dirección de a una fila y/o una columna de la posición  $(i, j)$ .

- a) Proponga un algoritmo que utilice la técnica de *Dividir y Conquistar* para encontrar las coordenadas del fondo (valor 0) de una *pirámide invertida*. El algoritmo debe tener complejidad  $O(\log n)$ .
- b) Justifique detalladamente la correctitud del algoritmo y su complejidad temporal, así como qué parte del algoritmo implementa cada una de las diferentes fases de la técnica.

Teodoro  
Freund 526/15 Orden ①

Eernab

① A.i

1	2	3
B	B	B

género manejador

Representación

manejador se representa con ostr

donde ostr es  $\text{tupla}(\text{arts: diccString}(\text{artestr}))$

donde artestr es  $\text{tupla}(\text{seccs: dicclog}(\text{Seccion, seccinfo}),$

ultVal: bool,

ultSecc:  $\text{tupla}(\text{secc: Seccion},$   
 $\text{it: itDicta}(\text{Seccion, seccinfo}))$

donde seccinfo es  $\text{ListaEnlazada}(\text{puntero}(\text{String}))$

[consideramos que String es  $\text{Soav}(\text{char})$ ]

Módulos utilizados y explicación de la estructura

- arts representa a los distintos artículos, que se guardan en un  $\text{diccString}$ , que es un diccionario con las claves de tipo  $\text{String}$  y, en este caso, los significados de artestr. Las operaciones de este módulo ~~más utilizadas~~ más utilizadas

(Obtener, Definir, Definido?) tienen complejidad

$O(\text{long}(\text{clave}))$ , pero por el enunciado, los nombres de los artículos están acotados por 99, es decir

que en nuestra estructura y en nuestro contexto estas operaciones serán constantes,  $O(99) = O(1)$

- `diccString` es un módulo que utiliza un Trie para acceder o buscar las claves (que son `String`), cada ~~nodo~~ eje  $\{$  representa una letra y en los nodos se almacenan los significados, es por esto que logramos las complejidades de  $O(\text{long}(\text{clave}))$
- `artesto` representa a cada artículo y guarda información de las secciones del mismo:
  - \* `secc` es un diccionario de `Seccion(Nat)` a `seccinfo` donde guarda la información de cada versión
  - \* `ultVal` es un `bool` que indica si el valor en `ultSecc` representa una sección válida
  - \* `ultSecc` guarda la última sección utilizada y un item dar a su pos. en el diccionario (`secc`) para acceder en  $O(1)$
- `dicclog` es un diccionario es un módulo que representa a un diccionario (donde las claves tienen un orden) con un AVL, esto permite trabajar con complejidades logarítmicas en la cantidad de elementos del `dicc`.
- `secc info` es una lista enlazada que almacena ~~p~~ punteros (`String`), es decir las distintas versiones de cada sección, de tal manera que el último elemento es la versión actual y el primero la inicial.



li Rep: estr  $\rightarrow$  bool

$Rep(e) \equiv true \Leftrightarrow$

- Todas las claves de los artículos tienen menos, ~~que~~ igual, que 90 caracteres

$$(\forall a: \text{Articulo}) (\text{Definido}(e.arts, a) \Rightarrow \text{Long}(a) \leq 90) \wedge$$

- Si tengo marcado que ultVal es true, entonces la información en ultSecc es válida

~~$$(\forall a: \text{Articulo}) (\text{Definido}(e.arts, a) \Rightarrow$$~~
~~$$(\text{Obtener}(e.arts, a).ultVal \Rightarrow$$~~
~~$$\text{Definido}(\text{Obtener}(e.arts, a)$$~~

$$(\forall a: \text{Articulo}) (\text{Definido}(e.arts, a) \Rightarrow \perp$$

$$((\forall b: \text{artestr}) (b = \text{Obtener}(e.arts, a) \Rightarrow \perp$$

$$(b.ultVal \Rightarrow (\text{Definido}(b.seccs, b.ultSecc.secc) \wedge$$

$$b.ultSecc.it = \text{CreatItSign}(b.seccs, b.ultSecc.secc))))))$$

- (\*) Dónde CreatItSign se piensa como CreatIt al diccionario, y Avanzar hasta que siguiente Clave sea igual a la clave pasada

$$\text{CreatItSign}(\text{dicc}(\alpha, \beta)^d \times \alpha \alpha) \rightarrow \text{itDicc}(\alpha, \beta)$$

$$\text{CreatItSign}(d, a) \equiv \text{CISRecu}(\text{CreatIt}(d), a)$$

$CISRecu : it_{Dice}(\alpha, \beta) \times \alpha \rightarrow it_{Dice}(\alpha, \beta)$

$CISRecu(it, \alpha) \equiv$  if  $\neg HorSigurate(it) \vee SigClave(it) = \alpha$  then

$\quad it$   
else

$\quad CISRecu(Avantat(it), \alpha)$   
Fi

① B.

Modificar Seccion (in/out e:estr, in art: articulo,  
in s: seccion, in txt: puntero(String) {

```

it ← CrearIt(vacio()) // it dice vacío, O(1)
def ← Obtener(e.arts, art) // ref a sign. del art. art O(10)=O(1)
def secc ← Vacio() // ref a un seccion, en ppo. vacío O(1)
if def.ultVal ∧ def.ultSecc.secc = s then // O(1)
    def ← def.ultSecc
    defsecc ← SiguienteSignificado(def.ultSecc.it) // O(1)
else
    def ← SiguienteSignificado(def.ultSecc.it) // O(1)
    defsecc ← Obtener(def.seccs, s) // por referencia, O(log(c))
    def.ultVal ← true // O(1)
    def.ultSecc.secc ← s // O(1)
    def.ultSecc.it ← CrearIt(def.seccs, defsecc) // O(log(c))
fi

```

Agregar Atms (defsecc, txt) // O(1)

} Si fue el último Accedido, la complejidad es O(1), sino  
O(log(c)), dependiendo de en que bloque del if entro

(\*) La operación CrearIt(dicc, clave) → itDicc  
es implementable ~~con~~ con complejidad logarítmica,  
por ejemplo, lo obtengo, lo elimino, y lo vuelvo a definir  
no la ~~defino~~ ya que no es un módulo ~~de~~ mi y  
implemento

se espera una implementación mejor que obtengo, borrar,  
definir, desde dentro del módulo

- ExisteArticulo( $e, z$ ) se fija si esta definido  $z$  en  $e.artls$ , esto es  $O(1)$
- AgregarArticulo( $e, z$ ) Define  $z$  en  $e.artls$  con significado  $\langle \text{Vacio}(), \text{false}, \langle 0, \text{CrearIt}(\text{Vacio}()) \rangle \rangle$   
esto es  $O(1)$

- ExisteSeccion( $e, z, s$ ) busca  $z$  en  $e.artls$  ( $O(1)$ ), y chequea si esta definida  $s$  en  $seccs$  ( $O(\log(c))$ )

$$O(1) + O(\log(c)) = O(\log(c))$$

- Versiones( $e, z, s$ ) busca  $z$  en  $e.artls$  ( $O(1)$ ), busca en  $seccs$  ( $O(\log(c))$ ) y pregunta por longitud( $seccinfo$ ) ( $O(1)$ ), y esa es la respuesta

$$O(1) + O(\log(c)) = O(\log(c))$$

- AgregarSeccion( $e, z, s, t$ ) busca  $z$  en  $e.artls$  ( $O(1)$ ), Define  $s$  en  $seccs$  con significado  $t$  ( $O(\log(c))$ )

Complejidad:  $O(\log(c))$

NOTAR QUE SE GENERA ALIASING CON EL TEXTO PASADO como REF.

Teodoro  
Fround 526/15

Orden 1

4

si fue  
el ult.  
visto

$O(1)$

• Modificar Seccion ( $e, z, s, t$ )

Busca  $z$  en  $e.arts$   $O(1)$ , busca la seccion ( $O(\log(c))$ )

Agrega Atras de la lista  $z$   $t$  ( $O(1)$ ), aatar que:

• el num de versiones Aumenta

• SE GENERA ALIASING, como En Agregar Seccion  
Comple:  $O(\log(c))$  o  $O(1)$  si fue el ult. mod.

• VerTexto( $e, z, s$ ) Busca  $z$  y  $s$  ( $O(\log(c))$ ) x Le

pide Ultimo a la lista ( $O(1)$ ), y devuelve  
el texto por ref.: HAY ALIASING, (se  
PODRIA COPIAR TAMBIEN)

comple:  $O(\log(c))$  o  $O(1)$  si fue el ultimo  
visto

• Inicio inicializa arts en vacio(),  $O(1)$

• Arrepentirse( $e, z, s, v$ ). Busca  $z$  y  $s$  ( $O(\log(c))$ ) o  $O(1)$

• Crea un iterador al primer elemento ( $O(1)$ )

• lo Avanza  $v-1$  veces ( $O(v)$ )

• LLAMA a modificar Seccion( $e, z, s, Sig(it)$ )  $O(1)$

PORQUE  $x$  ES EL  
ULT. modificado

☑ SE SUPONE QUE ARREPENTIRSE AUMENTA EN 1 LA  
~~VARIA~~ CANTIDAD DE VERSIONES

☑ NO ES CLARO SI VerTexto Modifica CUAL FUE  
EL ULTIMO ACCESADO, PERO ES IMPLEMENTADO Y  
NO ALTRA COMPLEJIDADES



(2) 2. B

Ordenar Mediciones (in/out A: arreglo[nat], in Limit, in limit){

```
B ← copiar(A) // O(n)
ultMax ← tan(A) - 1 // O(1)
for i ← tan(A) - 1 to 0 // O(1)
  if B[i] > H then // O(1)
    A[ultMax] ← B[i] // O(1)
    ultMax ← ultMax - 1 // O(1)
  fi
end for // O(n), rep. del ciclo

PrimMen ← 0 // O(1)
for i ← tan(A)-1 0 to tan(A) // O(1)
  if B[i] < L then // O(1)
    A[PrimMen] ← B[i] // O(1)
    PrimMen ← PrimMen + 1 // O(1)
  fi
end for // O(n)
```

```
while PrimMen ≤ ultMax do // O(1)
  min ← H+1 // O(1)
  minPos ← 0 // O(1)
  for i ← 0 to tan(B)-1 // O(1)
    if B[i] < min then // O(1)
      min ← B[i], minPos ← i // O(1)
    fi
  end for // O(n)
  B[i] ← H+2 // O(1)
  A[PrimMen] ← min // O(1)
  PrimMen ← PrimMen + 1 // O(1)
end while // O(1)
```

```

C ← vacío() // O(1)
for i ← 0 to tan(B) - 1 // O(1)
    if B[i] ≤ L ∧ B[i] ≥ L then // O(1)
        | Agregar Atras(C, B[i]) // O(1)
        | fi
    end for

```

$\{O(\log(n))\}$  Amortizado?  
 // agrego a la suma  $\log(n)$  elementos

```

SelectionSort(C) // O(log²(n))
i ← 0 // O(1)
while prinMen ≤ ultMax do // O(1)
    | A[prinMen + i] ← C[i] // O(1)
    | i ← i + 1 // O(1)
endwhile // O(log(n))

```

b. La idea del algoritmo es que primero busca los valores mayores que  $H$  y los inserto en el orden relativo en el que venían al final de  $A$  arreglado, Análogamente para los menores que  $L$ , esto me lleva  $O(n)$ , luego copio los valores dentro de  $[L, H]$ , que son a la suma  $\log(n)$ , y los ordeno con Selection Sort en  $O(\log^2(n))$  y al finalizar los inserto en orden.

Luego, la complejidad queda  $O(n + 2\log(n) + \log^2(n))$  pero sabemos que  $O(\log(n)) \subseteq O(n)$ , entonces tenemos  $O(n + \log^2(n))$

Como sabemos que  $O(\log(n)) \subseteq O(n)$ , sabemos que  $O(\log(\sqrt{n})) \subseteq O(\sqrt{n})$



Teodoro  
Freund 526/15

Orden 1

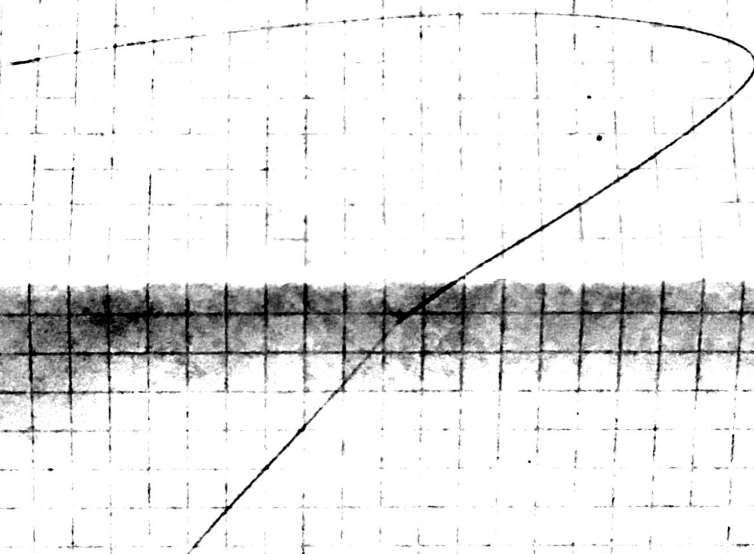
6

pero  ~~$O(\log(\sqrt{n}))$~~   $\log(\sqrt{n}) = \frac{1}{2} \log(n)$

entonces  $O(\log(\sqrt{n})) \leq O(\sqrt{n})$

Y se tiene que  $O(\log^2(n)) \leq O(\sqrt{n}^2) = O(n)$

Y así obtenemos la complejidad requerida



Teodoro  
Freund 526/15

Orden 1

7

③ 2, b

Encuentra Pto (in  $A: \text{arreglo}(\text{arreglo}(\text{nat}))$ )  $\rightarrow \text{res}: \langle \text{nat}, \text{nat} \rangle$   
 $\{ \text{res} \leftarrow \text{EncPtoAux}(A, 1, \text{tam}(A), 1, \text{tam}(A[1])) \}$

EncPtoAux (in  $A: \text{arreglo}(\text{arreglo}(\text{nat}))$ , in  $l_1: \text{nat}$ ,  
in  $h_1: \text{nat}$ , in  $l_2: \text{nat}$ , in  $h_2: \text{nat}$ )  $\rightarrow \text{res}: \langle \text{nat}, \text{nat} \rangle$

{ if  $l_1 = h_1 \wedge l_2 = h_2$  then //  $O(1)$   
|  $\text{res} \leftarrow \langle l_1, l_2 \rangle$  //  $O(1)$

else

|  $m_1 \leftarrow l_1 + \frac{h_1 - l_1}{2}$  //  $O(1)$

|  $m_2 \leftarrow l_2 + \frac{h_2 - l_2}{2}$  //  $O(1)$

| if  $A[l_1][m_1] \leq A[l_1][m_1 + 1]$  then //  $O(1)$

| | if  $A[m_2][l_1] \leq A[m_2 + 1][l_1]$  then //  $O(1)$

| |  $\text{res} \leftarrow \text{EncPtoAux}(A, l_1, m_1, l_2, m_2)$

| else

| |  $\text{res} \leftarrow \text{EncPtoAux}(A, l_1, m_1, m_2 + 1, h_2)$

| fi

| else

| | if  $A[m_2][l_1] \leq A[m_2 + 1][l_1]$  then //  $O(1)$

| |  $\text{res} \leftarrow \text{EncPtoAux}(A, m_1 + 1, h_1, l_2, m_2)$

| else

| |  $\text{res} \leftarrow \text{EncPtoAux}(A, m_1 + 1, h_1, m_2 + 1, h_2)$

| fi

fi

fi

}

La idea del algoritmo es que cualquier fila (o columna) que mire de cerca para el lado donde se encuentra el 0, es lo mismo que hacer Búsqueda Binaria en una fila cualquiera, y Búsqueda Binaria en una columna cualquiera.

Por otro lado, cada llamada a  $\text{EncPtoAux}$  tiene complejidad dada por:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{si no} \end{cases}$$

Donde es importante notar que la matriz la puedo dividir por una de tamaño  $2^k$ , para algún  $k$ , y dividirla siempre en submatrices de tamaño potencia de 2.

Donde es importante notar que la matriz es cuadrada, que el tamaño de la submatriz que proceso está dada por  $h_i - l_i + 1 = h_j - l_j + 1$ , es decir que la diferencia entre  $h_i$  y  $l_i$  se mantiene.

Luego, puedo aplicar el teorema maestro, donde  $a=1$ ,  $b=2$  y  $f(n) \in \Theta(1)$ .

Luego  $\log_2(1) = 0$ , entonces entro en el segundo caso, y a que  $f(n) \in \Theta(n^{\log_2(1)}) = \Theta(1)$ .

y obtengo que  $T(n) = \Theta(n^{\log_2(1)} \log(n)) = \Theta(\log(n))$ ,

la complejidad deseada.